

# Shared-Memory Parallelization Library (SMPlib)

*Version 2.5, 2001-05*

Haoqiang Jin <hjin@nas.nasa.gov>

*NASA Advanced Supercomputing Division, NASA Ames Research Center*

The Shared-Memory Parallelization Library (SMPlib) is designed for parallel processing on a shared memory machine. The concept was derived from MLPlib [1] developed by James Taft, but with extended functionality and syntax. The SMPlib routines are implemented with the standard Unix *fork*, *mmap* and *flock* (or more efficient atomic operation if available). The Application Programming Interface (API) of the library has been defined for both C and Fortran and is compatible with 32-bit and 64-bit hardware architectures. An overview of SMPlib and its comparison with other programming paradigms is given in [2]. This document summarizes the SMPlib v2.5 API.

## Contents

<b>1 Revision History</b>	<b>2</b>
<b>2 Environment Variables</b>	<b>3</b>
<b>3 Application Programming Interface</b>	<b>4</b>
3.1 Prototype Definition . . . . .	4
3.2 Process Creation . . . . .	4
3.3 Memory Allocation . . . . .	5
3.4 Synchronization . . . . .	7
3.5 Data Reduction . . . . .	9
3.6 Utility Routines . . . . .	9
<b>4 Non-SMPlib-API Routines</b>	<b>12</b>
<b>5 List of Error Codes</b>	<b>13</b>
<b>References</b>	<b>13</b>

## 1 Revision History

Changes in version 2.5 –

- new functions: `SMP_Barriern()` for numbered barrier to synchronize a subset of processes
- `SMP_Setoption()`/`SMP_Getoption()` to set/get SMPlib options
- Clarification on `SMP_Finish()` and `SMP_Join()` to allow the master process continue

Changes in version 2.4 –

- new functions: `SMP_Join()` to join forked processes,
- `SMP_Pinit_flag()` to set *pinit* flags from a call,
- `SMP_Version()` to return the SMPlib version number
- new environment variable: `SMP_DB_HOLD` to support debugger attachment
- change to environment variable: `SMP_PINIT` to support pinning step
- relax the restriction on the KIND type passing to the Fortran interface `SMP_GETSHMEMP`

Changes in version 2.3 –

- new functions: `SMP_Reduce()` for data reduction,
- `SMP_Resetshmem()` for shared-memory re-allocation
- new functions for error handling: `SMP_Perror()`, `SMP_Errcode()`, `SMP_Errorstring()`
- return value defined for: `SMP_Signal()` and `SMP_Wait()`

Changes in version 2.2 –

- new environment variable: `SMP_PINIT_OFFSET` to work with `SMP_PINIT`
- new functions: `SMP_Pinit()` and `SMP_Pinit_thread()` for the use in non-SMP programs

Changes in version 2.1 –

- new timer functions: `SMP_Timer_*`() to support timers
- new environment variable: `SMP_TLOG` to allow profiling SMPlib calls
- description on the environment variable: `SMP_PINIT`

Changes in version 2.0 –

- API changes to: `SMP_Fork()` for the consistent definition of `num_procs`,
- `SMP_Signal`/`SMP_Wait()` etc. for an additional `tid` argument

Changes in version 1.2 –

- new functions: `SMP_GETSHMEMP()` to the Fortran interface,
- `SMP_Forkthread()` to enable specification of threads

Changes in version 1.1 –

- function name change: `SMP_Getmem` to `SMP_Getshmem`
- function type change: `int SMP_Setlock/SMP_Unsetlock` to  
`void SMP_Setlock/SMP_Unsetlock`
- function argument change: `SMP_Signal(int node, int number)` to  
`SMP_Signal(int node)`
- new functions: `SMP_Testsignal`, `SMP_Setsignal`, `SMP_Testwait`, `SMP_Ackwait`,  
`SMP_Pagesize`

Version 1.0 –

- initial release

## 2 Environment Variables

**SMP\_NUM\_PROCS** - Number of processes to be forked.

Default uses the argument from `SMP_FORK()` or `SMP_FORKTHREAD()`.

**OMP\_NUM\_THREADS** - Number of threads per process.

Default uses the argument from `SMP_FORKTHREAD()` or 1 for `SMP_FORK()`.

**SMP\_TMPDIR** - Directory for temporary SMPlib files.

Default is “.” (the current working directory).

**SMP\_DEBUG** - Debugging flag, default is 0.

- 0 – no debugging information,
- 1 – debugging information printed to `<stdout>`,
- 2 – debugging information written to `smp_lib.log_myid`.

**SMP\_TLOG** - Time profiling flag, default is 0.

This flag is meaningful only if the TLOG option is compiled into SMPlib to perform time profiling of SMPlib calls.

**SMP\_PINIT** - The pin-to-node flag, default is 0.

This flag takes a format of “`pstart[:pstep]`” with a positive `pstart` to indicate the CPU start number for pinning and an optional `pstep` to indicate the CPU pinning step. Without `pstep` specified, a value of 1 is used. No pin-to-node is applied if `pstart` is less or equal to 0. A value of `pstart` larger than 1 is equivalent to a CPU offset number being specified.

This flag is meaningful for machines where processor pinning is implemented.

**SMP\_DB\_HOLD** - Hold processes after forking.

This flag allows debuggers to attach to the forked processes. By default, the flag is not set.

## 3 Application Programming Interface

SMPlib APIs for both C and Fortran languages are listed in this section. The Fortran interface has a very similar syntax as the C interface except for the memory allocation routines, such as `SMP_GETSHMEM` and `SMP_GETLOCMEM`. `SMP_GETSHMEMP` is added for a platform that supports the Cray pointer. So, in most cases both C and Fortran interfaces are described together. For those routines error conditions may exist, the corresponding error codes are included; otherwise, no error will be generated. A list of error codes is given in Section 5.

### 3.1 Prototype Definition

The prototypes and constants for most of the SMPlib routines are defined in

“`smp_lib.h`” (C interface)  
 “`smp_libf.h`” (Fortran interface).

The prototypes for the `SMP_Timer*` routines are defined in

“`smp_timer.h`” (C interface)  
 “`smp_timerf.h`” (Fortran interface).

### 3.2 Process Creation

```
void SMP_Init(void);
```

SUBROUTINE `SMP_INIT`

- initializes SMPlib parameters, should be the first thing (other than those setting options such as `SMP_Setoption`) to call in an SMP program. Environment variable `SMP_TMPDIR` is used to define the directory for temporary SMPlib files, default to “.” (the current working directory). Environment variable `SMP_DEBUG` is checked.

```
int SMP_Fork(int num_procs);
INTEGER FUNCTION SMP_FORK(NUM_PROCS)
INTEGER NUM_PROCS
```

- forks “`num_procs`” processes and return the rank (0 to `num_procs`-1) of the current process in the process group. If `num_procs`=0, the environment variable `SMP_NUM_PROCS` will be used. If `SMP_NUM_PROCS` is undefined, no subprocess is created, i.e. `num_procs`=1 is used. Number of threads per process is defined by the environment variable `OMP_NUM_THREADS`. If `OMP_NUM_THREADS` is undefined, one (1) is assumed.
- error code: `SMP_ERR_NPROC`, `SMP_ERR_NTHR`

```
int SMP_Forkthread(int num_procs, int num_threads[]);
INTEGER FUNCTION SMP_FORKTHREAD(NUM_PROCS, NUM_THREADS)
INTEGER NUM_PROCS, NUM_THREADS(0:*)
```

- forks “`num_procs`” processes and return the rank (0 to `num_procs`-1) of the current process in the process group. If `num_procs`=0, the environment variable `SMP_NUM_PROCS` will be used. If `SMP_NUM_PROCS` is undefined, no subprocess is created, i.e. `num_procs`=1 is used. In addition, `num_threads[i]` of threads is specified for process `i`. If `num_threads[i]` is 0, the variable `OMP_NUM_THREADS` will be checked; if `OMP_NUM_THREADS` is undefined, one (1) is assumed.

- error code: `SMP_ERR_NPROC`, `SMP_ERR_NTHR`

```
void SMP_Finish(void);
SUBROUTINE SMP_FINISH
```

- finishes and cleans up things, should be called by all processes. Any previously allocated shared memory buffers will be freed at this point. All subprocesses perform a *join* function and only the master process may continue execution after `SMP_Finish()`. Use of this function is important for cases where only the master process is allowed for further execution, such as, when mixing SMP with MPI.

```
void SMP_Join(void);
SUBROUTINE SMP_JOIN
```

- joins all the forked processes, should be called by all processes. This function is very similar to `SMP_Finish()` except that it does not free any previously allocated shared memory buffers. Only the master process may continue execution after `SMP_Join()`.

### 3.3 Memory Allocation

```
void *SMP_Getshmem(size_t size);
```

- C: gets a piece of shared memory with “`size`” bytes. A pointer to the allocated memory segment is returned.

*Restriction:* The function is usually called before `SMP_Fork`. But the function may also be used after `SMP_Fork`, in which case it must be called by all the processes with the same requested size. The same restriction applies to other shared-memory-allocation routines (including Fortran routines).

- error code: `SMP_ERR_MALLOC`

```
SUBROUTINE SMP_GETSHMEM(REFP, KIND, SIZE, IOFF)
<type>  REFP(1)
INTEGER KIND, SIZE
<pointer> IOFF
```

- Fortran: gets a piece of shared memory for “`kind`” with “`size`” elements.

<code>&lt;type&gt;</code>	is the reference type, such as <code>REAL</code> , <code>INTEGER</code> ,...
<code>KIND</code>	is one of (1,2,4,8,16) as the size of the reference type
<code>SIZE</code>	is the number of elements to be allocated
<code>&lt;pointer&gt;</code>	is a type large enough to hold an address: for 32-bit, this is <code>INTEGER*4</code> ; for 64-bit, this is <code>INTEGER*8</code> .

Reference to the allocated shared memory is done by `REFP(IOFF)` as the first element. The function may be used after `SMP_FORK`, but should be called by all the processes with the same requested size.

- error code: `SMP_ERR_MALLOC`, `SMP_ERR_KIND`

```
SUBROUTINE SMP_GETSHMEMP(KIND, SIZE, IPTR)
INTEGER KIND, SIZE
POINTER IPTR
```

- Fortran: gets a piece of shared memory for “kind” with “size” elements.

`KIND` is a positive scaling factor for the size of the data type

`SIZE` is the number of elements to be allocated

`IPTR` is the returned pointer that points to the allocated memory.

This function can be used on platform that supports the Cray pointer. Typically a variable is declared as

```
<type> VAR(100)
POINTER (IPTR,VAR)
```

Reference to the allocated shared memory is done by `VAR(1)` as the first element. The approach works for both 32-bit and 64-bit platforms. Combining with a large value of “kind” this function can be used to allocate memory larger than 2 GB. See `SMP_Getshmem()` for usage restriction.

- error code: `SMP_ERR_MALLOC`, `SMP_ERR_KIND`

```
void *SMP_Resetshmem(void *ptr, size_t size);
```

- C: resets the size of a shared memory buffer that was previously allocated. “`ptr`” points to the old memory buffer and “`size`” is the new size. The function returns a pointer to the newly re-allocated memory buffer. If `size=0`, the function frees the previously allocated shared memory buffer. The same restriction to `SMP_Getshmem` applies to this function as well. See `SMP_Getshmem()` for usage restriction.

- error code: `SMP_ERR_MALLOC`, `SMP_ERR_MEMPTR`

```
SUBROUTINE SMP_RESETSHMEM(REFP, KIND, SIZE, IOFF)
<type> REFP(1)
INTEGER KIND, SIZE
<pointer> IOFF
```

- Fortran: resets the size of a shared memory buffer that was previously allocated. See `SMP_GETSHMEM` for a description on the argument list and `SMP_Getshmem()` for usage restriction.

- error code: `SMP_ERR_MALLOC`, `SMP_ERR_MEMPTR`, `SMP_ERR_KIND`

```
SUBROUTINE SMP_RESETSHMEMP(KIND, SIZE, IPTR)
INTEGER KIND, SIZE
POINTER IPTR
```

- Fortran: resets the size of a shared memory buffer that was previously allocated. See `SMP_GETSHMEMP` for a description on the argument list and `SMP_Getshmem()` for usage restriction.
- error code: `SMP_ERR_MALLOC`, `SMP_ERR_MEMPTR`, `SMP_ERR_KIND`

```
int SMP_Pagesize(void);
INTEGER FUNCTION SMP_PAGESIZE()
```

- returns the memory page size in bytes.

### 3.4 Synchronization

```
void SMP_Barrier(void);
SUBROUTINE SMP_BARRIER
```

- synchronizes all processes and has to be called by each process.

```
void SMP_Barriern(int b_no, int b_numprocs, int b_procid);
SUBROUTINE SMP_BARRIERN(B_NO, B_NUMPROCS, B_PROCID)
INTEGER B_NO, B_NUMPROCS, B_PROCID
```

- synchronizes a subset of processes with a numbered barrier. `b_no` is the barrier number, ranging from 0 to `MAX_NBARRIER-1`, `b_numprocs` is the number of processes involved in the numbered barrier, and `b_procid` (between 0 and `b_numprocs-1`) is a virtual process id used in this barrier. This function must be called with the same `b_no` and `b_numprocs` by those processes involved in the barrier, and the virtual process id (`b_procid`) must be uniquely defined within the subset. `MAX_NBARRIER` ( $\geq 4$ ) is implementation defined.

- error code: `SMP_ERR_NPROC`, `SMP_ERR_NODE`

```
int SMP_Testsignal(int node, int tid);
INTEGER FUNCTION SMP_TESTSIGNAL(NODE, TID)
INTEGER NODE, TID
```

- tests if a node is ready for signal from the current node. If `node < 0`, signal to any node. Returns the node number the signal will actually be sent to. The call ensures any previous signal sent to node from the current node has been taken. The second argument `tid` tags the signal to a particular thread.

- error code: `SMP_ERR_NODE`, `SMP_ERR_TID`

```
void SMP_SetSignal(int node, int tid);
SUBROUTINE SMP_SETSIGNAL(NODE, TID)
INTEGER NODE, TID
```

- sets a signal for node after `SMP_Testsignal` is called. The second argument `tid` tags the signal to a particular thread.

- error code: `SMP_ERR_NODE`, `SMP_ERR_TID`, `SMP_ERR_MNODE`

```
int SMP_Signal(int node, int tid);
INTEGER FUNCTION SMP_SIGNAL(NODE, TID)
INTEGER NODE, TID
```

- sends a signal to node. If `node < 0`, signal to any node. Returns the node number the signal will actually be sent to. The second argument `tid` tags the signal to a particular thread. This function is equivalent to `SMP_Testsignal + SMP_SetSignal`.

- error code: `SMP_ERR_NODE`, `SMP_ERR_TID`

```
int SMP_Testwait(int node, int tid);
INTEGER FUNCTION SMP_TESTWAIT(NODE, TID)
INTEGER NODE, TID
```

- waits for a signal from node. If `node < 0`, signal from any node. Returns the node number the signal is actually from. The second argument `tid` tags the signal to a particular thread.

- error code: `SMP_ERR_NODE`, `SMP_ERR_TID`

```
void SMP_Ackwait(int node, int tid);
SUBROUTINE SMP_ACKWAIT(NODE, TID)
INTEGER NODE, TID
```

- acknowledges the reception of a signal from node after `SMP_Testwait` is called. The second argument `tid` tags the signal to a particular thread.

- error code: `SMP_ERR_NODE`, `SMP_ERR_TID`, `SMP_ERR_MNODE`

```
int SMP_Wait(int node, int tid);
INTEGER FUNCTION SMP_WAIT(NODE, TID)
INTEGER NODE, TID
```

- waits for a signal from node. If `node < 0`, signal from any node. Returns the node number the signal is actually from. The second argument `tid` tags the signal to a particular thread. This function is equivalent to `SMP_Testwait + SMP_Ackwait`.

- error code: `SMP_ERR_NODE`, `SMP_ERR_TID`

```
void SMP_Setlock(void);
SUBROUTINE SMP_SETLOCK
```

- waits for the SMPlib global lock and set the lock when it is available.

```
void SMP_Unsetlock(void);
SUBROUTINE SMP_UNSETLOCK
```

- releases the SMPlib global lock after it was set.

### 3.5 Data Reduction

```
void SMP_Reduce(void *data_in, void *data_out, int nitems,
                 int dtype, int opr, int node);

SUBROUTINE SMP_REDUCE(DATA_IN, DATA_OUT, NITEMS, DTYPY, OPR, NODE)
<type>  DATA_IN(*), DATA_OUT(*)
INTEGER NITEMS, DTYPY, OPR, NODE
```

- performs data reduction among processes.

**data\_in** - points to the input data set  
**data\_out** - points to the output data set  
**nitems** - number of items in the data set  
**dtype** - data type for input/output data sets  
**opr** - reduction operator  
**node** - node to which data are reduced (-1 for all nodes)

**<type>** should match with the size specified by **dtype**.

Data type (**dtype**) is one of (Fortran or C):

<b>SMP_DT_INT1</b>	- integer*1	or char
<b>SMP_DT_INT2</b>	- integer*2	or short
<b>SMP_DT_INT4</b>	- integer*4	or int
<b>SMP_DT_INT8</b>	- integer*8	or long (or long long)
<b>SMP_DT_REAL4</b>	- real*4	or float
<b>SMP_DT_REAL8</b>	- real*8	or double
<b>SMP_DT_CMPLX8</b>	- complex*8	or struct{float r, i;}
<b>SMP_DT_CMPLX16</b>	- complex*16	or struct{double r, i;}

Reduction operator (**opr**) is one of (with the supported data types):

<b>SMP_OPR_SUM</b>	- sum	(all)
<b>SMP_OPR_PROD</b>	- product	(all)
<b>SMP_OPR_MIN</b>	- minimum	(INT*, REAL*)
<b>SMP_OPR_MAX</b>	- maximum	(INT*, REAL*)
<b>SMP_OPR_BAND</b>	- logical and	(INT*)
<b>SMP_OPR_BAND</b>	- bitwise and	(INT*)
<b>SMP_OPR_LOR</b>	- logical or	(INT*)
<b>SMP_OPR_BOR</b>	- bitwise or	(INT*)
<b>SMP_OPR_LXOR</b>	- logical xor	(INT*)
<b>SMP_OPR_BXOR</b>	- bitwise xor	(INT*)

- error code: **SMP\_ERR\_DTYPE**, **SMP\_ERR\_RDOPR**, **SMP\_ERR\_MIXED**

### 3.6 Utility Routines

```
int SMP_Debug(int newflag);

INTEGER FUNCTION SMP_DEBUG(NEWFLAG)
INTEGER NEWFLAG
```

- resets the debug flag (0, 1 or 2) and returns the previous value. This function overwrites the value defined by the environment variable `SMP_DEBUG`.

```
int SMP_Errcode(void);
INTEGER FUNCTION SMP_ERRCODE()
```

- returns the current error code (see Section 5 for a list).

```
char *SMP_ErrorString(int errcode);
SUBROUTINE SMP_ERRSTRING(ERRCODE, ERRSTR)
INTEGER   ERRCODE
CHARACTER ERRSTR*(*)
```

- returns a string defined for `errcode` (see Section 5 for a list). The Fortran version copies the string to `ERRSTR`.

```
int SMP_Getoption(char *sopt, char *svalue, int lvalue);
SUBROUTINE SMP_GETOPTION(SOPT, SVALUE, IFLAG)
CHARACTER SOPT*(*), SVALUE*(*)
INTEGER   IFLAG
```

- gets option value(s) for a given SMPlib option (`sopt`). Value(s) is returned as a string into `svalue` (with a maximum size of `lvalue` in C). The function returns a positive flag (to `IFLAG` in Fortran) on success and a non-positive ( $\leq 0$ ) value on failure. A list of supported options is implementation dependent, but see `SMP_Setoption` for minimum requirements.

- error code: `SMP_ERR_OPTION`, `SMP_ERR_MALLOC`

```
int SMP_Myid(void);
INTEGER FUNCTION SMP_MYID()
```

- returns the rank of the current process.

```
int SMP_Numprocs(void);
INTEGER FUNCTION SMP_NUMPROCS()
```

- returns the number of processes, including the master.

```
void SMP_Perror(char *msg);
SUBROUTINE SMP_PERROR(MSG)
CHARACTER MSG*(*)
```

- prints the current error message to `<stderr>`. If the current error condition (`errcode`) is `SMP_NO_ERR`, no message will be printed. If `msg` is given, a prefix “`msg:`” will be included.

```
void SMP_Pinit_flag(int pinit, int pinit_step, int flag);
SUBROUTINE SMP_PINIT_FLAG(PINIT, PINIT_STEP, FLAG)
INTEGER PINIT, PINIT_STEP, FLAG
```

- sets the *pinit* flags beside those from the environment variable **SMP\_PINIT**. If **flag** > 0, this function overwrites those values defined by **SMP\_PINIT**; if **flag** ≤ 0, the function adds offsets to the corresponding values defined by **SMP\_PINIT**.

In order to have the desired effect, **SMP\_Pinit\_flag()** must be called before **SMP\_Fork()**, **SMP\_Forkthread()**, **SMP\_Pinit()**, or **SMP\_Pinit\_thread()** (the last two are non-API functions described in Section 4).

```
int SMP_Setoption(char *sopt, char *svalue);
SUBROUTINE SMP_SETOPTION(SOPT, SVALUE, IFLAG)
CHARACTER SOPT*(*) , SVALUE*(*)
INTEGER     IFLAG
```

- sets option value(s) for a given SMPlib option (**sopt**). Value(s) is given as a string in **svalue** and multiple values are colon (:) separated. The function returns a positive flag (to **IFLAG** in Fortran) on success and a non-positive (≤ 0) value on failure. The function is intended to replace other option setting functions (such as **SMP\_Debug**, **SMP\_Pinit\_flag**). A list of supported options is implementation dependent, but to the minimum, the following options are defined:

```
sopt="DEBUG"   – the debug flag
sopt="PINIT"   – the pinit flags "<pinit>[:<pinit_step>:<flag>]".
```

- error code: **SMP\_ERR\_OPTION**, **SMP\_ERR\_VALUE**, **SMP\_ERR\_MALLOC**

```
void SMP_Version(char **verno, char **vdate);
SUBROUTINE SMP_VERSION(VERNO, VDATE)
CHARACTER VERNO*(*) , VDATE*(*)
```

- returns the SMPlib version number and date.

```
double SMP_Wtime(void);
DOUBLE PRECISION FUNCTION SMP_WTIME()
```

- gets wallclock time in seconds.

.....  
The following routines define a set of timers besides the **SMP\_Wtime()** function. Maximum number of timers is 64. The “timer” field is from 1 to 64. The timer routines are not thread-safe.

```
void SMP_Timer_init(int timer);
SUBROUTINE SMP_TIMER_INIT(TIMER)
INTEGER TIMER
```

- initializes a timer (if **timer** > 0) or all timers (if **timer==0**). **SMP\_Init()** initializes all timers. This function is used to reset a timer (to zero).
- error code: **SMP\_ERR\_TIMER**

```

void SMP_Timer_start(int timer);

SUBROUTINE SMP_TIMER_START(TIMER)
INTEGER TIMER

    – starts a timer.

    – error code: SMP_ERR_TIMER, SMP_ERR_TSTOP

void SMP_Timer_stop(int timer);

SUBROUTINE SMP_TIMER_STOP(TIMER)
INTEGER TIMER

    – stops a timer.

    – error code: SMP_ERR_TIMER, SMP_ERR_TSTART

double SMP_Timer_read(int timer);

DOUBLE PRECISION FUNCTION SMP_TIMER_READ(TIMER)
INTEGER TIMER

    – returns the current value of a timer in seconds. The timer will be stopped first if it has not.

    – error code: SMP_ERR_TIMER

void SMP_Timer_string(int timer, char *str);

SUBROUTINE SMP_TIMER_STRING(TIMER, STR)
INTEGER TIMER
CHARACTER STR*(*)

    – defines a string for a timer. The string is used by SMP_Timer_print().

    – error code: SMP_ERR_TIMER

void SMP_Timer_print(int timer);

SUBROUTINE SMP_TIMER_PRINT(TIMER)
INTEGER TIMER

    – prints a timer (if timer > 0) or all timers (if timer==0).

    – error code: SMP_ERR_TIMER

```

## 4 Non-SMPlib-API Routines

The next two routines are included in v2.2+, but they are not part of the SMPlib v2 API. The two routines are intended for calling by non-SMP programs.

```

void SMP_Pinit(int nprocs, int myid);

SUBROUTINE SMP_PINIT(NPROCS, MYID)
INTEGER NPROCS, MYID

```

- performs *pin-to-node* for process MYID ( $0 \dots nprocs - 1$ ), assuming a fixed number of threads per process (defined by `OMP_NUM_THREADS`). The environment variable `SMP_PINIT` is still checked for the actual activation. Note: this function should NOT be used together with `SMP_Fork()` or `SMP_Forkthread()`.
- error code: `SMP_ERR_NODE`, `SMP_ERR_PINIT`

```
void SMP_Pinit_thread(int nprocs, int myid, int nthreads[]);

SUBROUTINE SMP_PINIT_THREAD(NPROCS, MYID, NTHREADS)
INTEGER NPROCS, MYID, NTHREADS(0:*)
```

- same as `SMP_Pinit` except that it accepts a variable number of threads per process. The same restrictions for `SMP_Pinit` apply to this function.
- error code: `SMP_ERR_NODE`, `SMP_ERR_PINIT`

## 5 List of Error Codes

<i>error code</i>	<i>value</i>	<i>error string</i>
<code>SMP_NO_ERR</code>	0	no error
<code>SMP_ERR_RDOOPR</code>	1	unknown reduction operator
<code>SMP_ERR_DTYPE</code>	2	unknown data type
<code>SMP_ERR_MIXED</code>	3	incorrect mix of data and reduction types
<code>SMP_ERR_MEMPTR</code>	4	invalid memory pointer
<code>SMP_ERR_MALLOC</code>	5	error in memory allocation
<code>SMP_ERR_NODE</code>	6	invalid node/proc number
<code>SMP_ERR_TID</code>	7	invalid thread number
<code>SMP_ERR_MNODE</code>	8	mismatched node numbers
<code>SMP_ERR_NPROC</code>	9	number of nodes exceeded limit
<code>SMP_ERR_NTHR</code>	10	number of threads exceeded limit
<code>SMP_ERR_TIMER</code>	11	timer out of range
<code>SMP_ERR_TSTART</code>	12	timer not yet started
<code>SMP_ERR_TSTOP</code>	13	timer not yet stopped
<code>SMP_ERR_KIND</code>	14	incorrect data kind
<code>SMP_ERR_PINIT</code>	15	pin-to-node not supported
<code>SMP_ERR_OPTION</code>	16	option not supported
<code>SMP_ERR_VALUE</code>	17	error reading option value

## References

- [1] J. Taft, “Achieving 60 GFLOP/s on the Production CFD Code OVERFLOW-MLP.” *Parallel Computing*, 27 (2001) 521.
- [2] H. Jin, G. Jost, “Performance Evaluation of Remote Memory Access Programming on Shared Memory Parallel Computers.” *NAS Technical Report* NAS-03-001, NASA Ames Research Center, Moffett Field, CA, 2003.